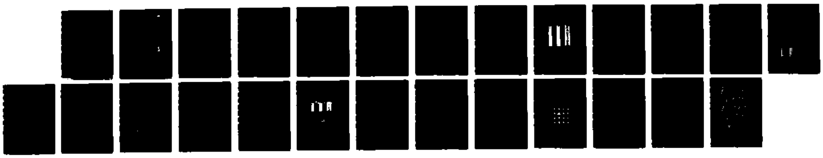
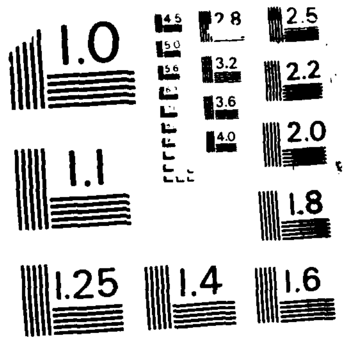
AD-A190 630   PROGRAMMING ENVIRONMENTS FOR SYSTOLIC ARRAYS(U)   1/1
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE
L SNYDER FEB 86 TR-86-02-02 N00014-85-K-0328
UNCLASSIFIED   F/G 12/5   NL




1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

AD-A190 630

(4)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Programming Environments for Systolic Arrays		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Lawrence Snyder		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0328
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science, FR-35 Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE February 1986
		13. NUMBER OF PAGES 22
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel programming, systolic arrays, Poker Programming Environment, parallel algorithms, parallel programming environments

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Although a systolic array is often thought of as a "hard wired" device, there are many reasons to want to program systolic algorithms. In this paper the problem of providing an efficacious programming environment is addressed. The difficulties of programming complex parallel algorithms are shown to be reduced by using a new concept of a parallel "program" which maximizes the use of graphical abstractions and minimizes the need for symbolic text. This concept is illustrated by the Poker Parallel Programming Environment which, although designed for a broader class of algorithms, illustrates the main

(continued)

88 1 12 197

Programming Environments for Systolic Arrays

Lawrence Snyder
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

TR 86-02-02

This document has been funded in part by the Office of Naval Research Contract No. N00014-85-K-0328 and the National Science Foundation Grant No. DCR-8416878.

Programming Environments for Systolic Arrays

Lawrence Snyder
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

Abstract

Although a systolic array is often thought of as a "hard wired" device, there are many reasons to want to program systolic algorithms. In this paper the problem of providing an efficacious programming environment is addressed. The difficulties of programming complex parallel algorithms are shown to be reduced by using a new concept of a parallel "program" which maximizes the use of graphical abstractions and minimizes the need for symbolic text. This concept is illustrated by the Poker Parallel Programming Environment which, although designed for a broader class of algorithms, illustrates the main features that a programming environment specialized to systolic computation should have.

Introduction

A programming environment is an integrated suite of software tools supporting all activities associated with writing and running programs. These activities include editing, compiling, loading, executing and debugging, as well as file manipulation, library access and various job control operations. In the event that the programming environment is for systolic arrays, the programs are parallel and thus impose additional demands on the system such as data formatting, management of multiple process sets, and the specification of data routings. Providing all of these facilities in a single integrated system requires a new concept of "parallel program," which, although it is quite different from FORTRAN and PASCAL, is nevertheless easier to use for a systolic array than a conventional programming language. The key to achieving this simplicity is to use graphics so extensively that the resulting programs appear to be dynamic versions of the diagrams presented in textbooks. These concepts, demonstrated in the Poker Parallel Programming Environment [1], will be described fully in this paper.

The Need for a Programming Environment

To many, a systolic array is a special purpose parallel computer that implements a single systolic algorithm in hardware. One might reasonably ask, therefore, what is there to program, why is a programming environment needed for systolic arrays? There are a variety reasons for wanting to program systolic algorithms.

If the intent is to build a systolic array in hardware, then one might begin by programming it to make sure that the design is functionally correct. This not only can be done easily in a programming environment such as Poker, but the environment can serve as a design tool to support the more detailed levels of the design. To begin, the algorithm is defined with the processor elements (PEs) executing simple arithmetic operations such as

$$c := c + ab$$

on full word data. Once this program is debugged and tested on various data sets, the program can be refined. Specifically, the arithmetic expressions can be replaced by equivalent functions expressed in more primitive operations, say by a register transfer level specification. The programming environment doesn't actually have special register transfer instructions; one simply programs them in the high level language. Structures like registers are implemented with Boolean vectors. Proceeding in this manner one moves to more and more detailed levels, for example, the logic level. The benefit – the usual benefit with stepwise refinement – is that local changes to a working program produce a working program that must match its predecessor(s) on the test data.[1]

The other users motivated to employ programming environments capable of supporting systolic computation are more obvious: users of programmable systolic arrays [3, 4], users of general purpose parallel computers choosing to use a systolic algorithm, or researchers designing new systolic algorithms. These groups, distinguished largely by the characteristics of the target machine, will be more interested in such operations as systolic algorithm composition than the stepwise refinement operation mentioned above.

Background

The Poker Parallel Programming Environment is a general programming environment for nonshared memory parallel computers, and thus is not specifically designed for systolic arrays. Poker is presented here as a system exhibiting many of the characteristics that one would expect of a programming environment specially built for systolic computation. The only potential harm is that Poker may be somewhat more general than is necessary for programming situations limited strictly to systolic algorithms. The only true instance of this

[1]This idea has been proposed as a basis of a full VLSI design system.[2]

problem is synchronization: Poker takes asynchronous communication as the default, while systolic arrays are synchronous. A Poker-like system specialized to systolic computation would be synchronous and might limit the generality of Poker in other ways. Surprisingly, there will be occasion in the paper to use most of the features of Poker, suggesting that systolic algorithms exhibit much of the diversity of general parallel algorithms.

Originally designed as a research tool for algorithm development for the Configurable Highly Parallel (CHiP) Computer [5], and for program development for the CHiP Computer's prototype, the Pringle Parallel Computer [6], Poker is presently being ported to the Cosmic Cube [7]. Design work on Poker was begun (at Purdue University) in January, 1982. The rudiments of the unstable system were exhibited in October 1982. The system, dramatically extended and enhanced, was completed for distribution (at the University of Washington) in October, 1985. Poker is written in C, runs on such machines as the VAX^{TM} 11/780 and cross compiles for, or emulates parallel machines. Thus, Poker is a sequential program and does not assume specific (or any) parallel hardware.

The Form of a Poker Program

Because the Poker environment has several nonstandard features, it is best to begin the presentation with a specific example. We choose the Kung-Leiserson band-matrix multiplication algorithm [8]. Recall that the algorithm accepts two band matrices, A and B of band-width u_1 and u_2 and produces a band matrix C of width $w = u_1 + u_2 - 1$ which is their product. We seek a direct translation.

Figures 1-5 show the constituent parts of the Poker program for the algorithm: *communication structure definition, process definition, process assignment, port name assignment,* and *stream name assignment*. Notice that of the five constituent parts, one is textual, one is tabular and three are graphical. Each component will be discussed in turn.

Communication Structure. The communication structure specifies the channels to be used by the processor elements (shown as boxes in Figure 1) for interprocessor communication. (Circles [1] should be ignored here.) If the programming environment is supporting a particular physical machine with a specific communication structure, then the graph describing this structure is programmed into the system once and is the basis of all programs. If the environment is used for a research tool or for a configurable general purpose computer [5] then the programmer will draw the interconnection structure by connecting the boxes with lines.

The lines are bidirectional data paths across which values can be transmitted. The squares at the ends of the perimeter edges are called *I/O pads* and represent places where

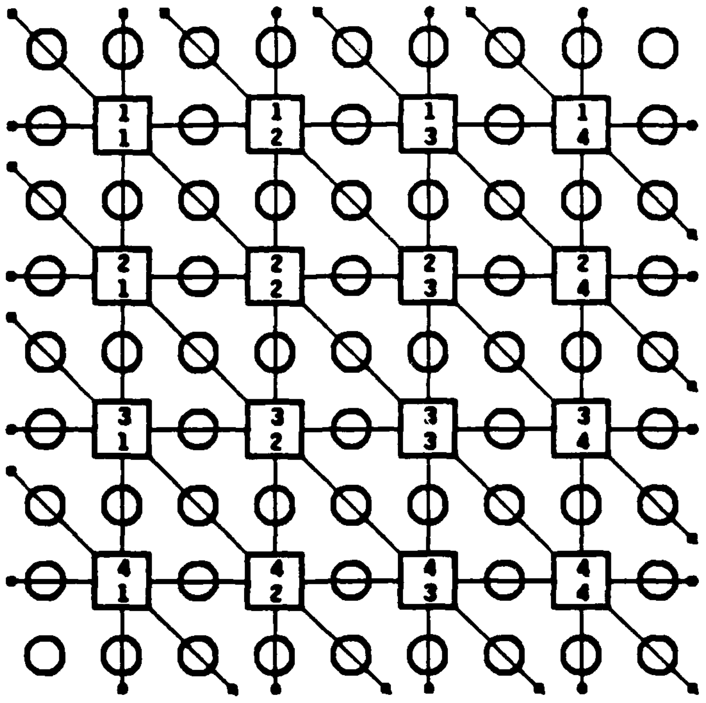


Figure 1. Communication structure

```
code inner;
trace a,b,c;
ports Ain, Bin, Cin,
   Aout, Bout, Cout;
begin
  real a,b,c;
       a := 0.0;
       b := 0.0;
       c := 0.0;
       while TRUE do
         begin
         Aout <- a;
         Bout <- b;
         Cout <- c;
         a <- Ain;
         b <- Bin;
         c <- Cin;
         c := c + a*b;
         end
end.
```

Figure 2. Process definition

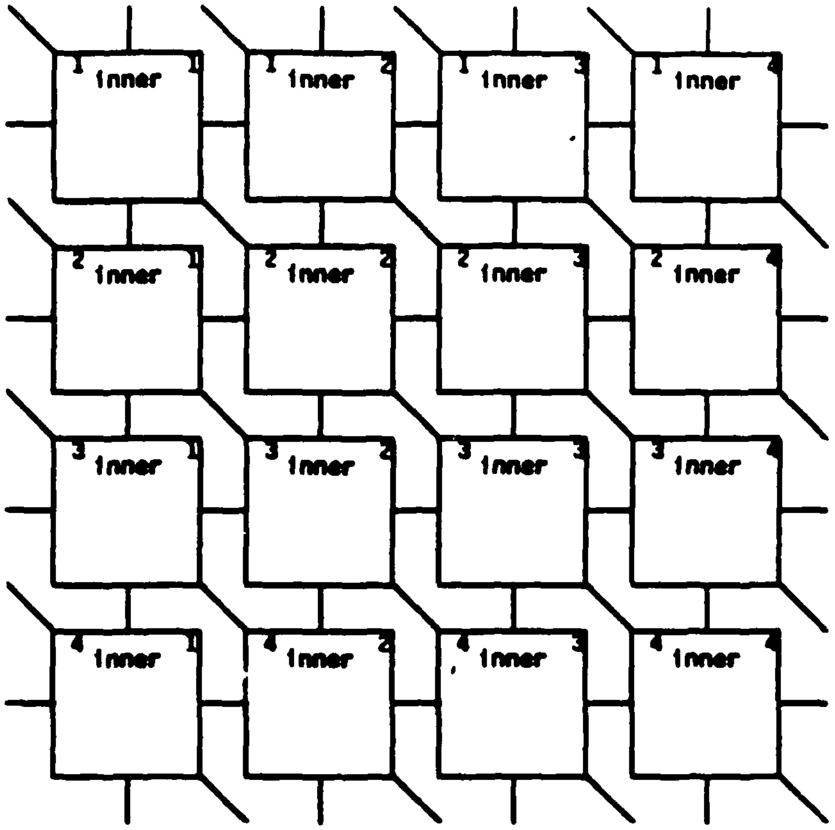


Figure 3. Process assignment mapping

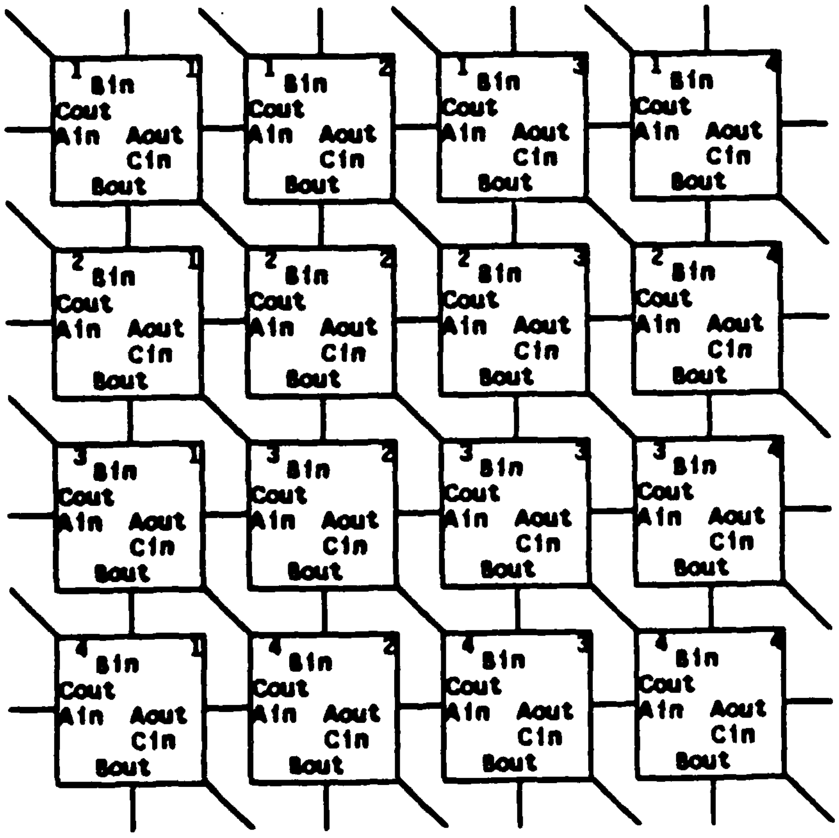


Figure 4. Port name assignment

STREAM				DESTINATION				
PAD	NAME	INDEX	DIR.	PORT NAME	DIRECTION	CODE NAME	I	J
1	Barrayin	1	input	bin	north	inner	1	1
2	Carrayout	5	output	cout	northwest	inner	1	2
3	Barrayin	2	input	bin	north	inner	1	2
4	Carrayout	6	output	cout	northwest	inner	1	3
5	Barrayin	3	input	bin	north	inner	1	3
6	Carrayout	7	output	cout	northwest	inner	1	4
7	Barrayin	4	input	bin	north	inner	1	4
8	Aarrayout	1	output	aout	east	inner	1	4
9	Carrayin	7	input	cin	southeast	inner	1	4
10	Aarrayout	2	output	aout	east	inner	2	4
11	Carrayin	6	input	cin	southeast	inner	2	4
12	Aarrayout	3	output	aout	east	inner	3	4
13	Carrayin	5	input	cin	southeast	inner	3	4
14	Aarrayout	4	output	aout	east	inner	4	4
15	Carrayin	4	input	cin	southeast	inner	4	4
16	Barrayout	4	output	bout	south	inner	4	4
17	Carrayin	3	input	cin	southeast	inner	4	3
18	Barrayout	3	output	bout	south	inner	4	3
19	Carrayin	2	input	cin	southeast	inner	4	2
20	Barrayout	2	output	bout	south	inner	4	2
21	Carrayin	1	input	cin	southeast	inner	4	1
22	Barrayout	1	output	bout	south	inner	4	1
23	Aarrayin	4	input	ain	west	inner	4	1
24	Carrayout	1	output	cout	northwest	inner	4	1
25	Aarrayin	3	input	ain	west	inner	3	1
26	Carrayout	2	output	cout	northwest	inner	3	1
27	Aarrayin	2	input	ain	west	inner	2	1
28	Carrayout	3	output	cout	northwest	inner	2	1
29	Aarrayin	1	input	ain	west	inner	1	1
30	Carrayout	4	output	cout	northwest	inner	1	1

Figure 5. Stream name assignment

data streams will enter the array. Notice that except for a 45° counter clockwise rotation, this diagram is an undirected version of the diagram used by Kung and Leiserson [8].

Process Definition. Each processor element of the array executes an inner product step for the matrix multiplication algorithm. (See Figure 2.) This is specified in a PASCAL-like language called *XX* (Dos Equis). The process repeatedly computes the inner product operation on scalar floating point values. Notice that the preamble of the program specifies the process name, *inner*, a set of variables to be traced and six ports. Ports are simply the names of the data paths connected to a processor element. Of special interest are the statements of the form

```
variable <- portname
portname <- variable
```

The former is an input command requesting that data be read from port *portname* and assigned to the variable. The second statement is an output command. The I/O is performed in a data driven fashion: Write statements transmit immediately unless doing so would overflow the recipients buffer; read statements execute immediately unless doing so would read an empty buffer; in both cases the processors block until the inhibiting condition is cleared. Issues of synchronization are discussed below.

Process Assignment. Although one often thinks of systolic arrays as SIMD architectures, that is, all processors execute the same instruction stream, it is not always the case [8]. So, to permit different processors to be assigned different operations, the process assignment specification is provided. As it happens each processor, in this example, executes the same code, (See Figure 3). Later, modification of the program will cause some processes to execute different codes. In addition, each process can be assigned a small set of parameters which permits further particularization. Notice that the graphic display shows the interconnections given in the communication structure definition; this is also true for the port naming display.

Port Name Assignment. The port naming facilities permit the port names used in the program specification to be associated with specific data paths. (See Figure 4.) Although it is typical for array type systolic algorithms to have a consistent port labelling that would permit one to label just a single generic cell rather than the whole array, systolic algorithms based on other graphs such as trees are not so easily labeled [5].

Stream Name Assignment. The data entering the array is assumed to be streams, which are arranged into files. The user specifies symbolic names and indexes for all of the I/O pads at the periphery of the array. (See Figure 5.) The A array, composed of four streams named *Aarray.1* through *Aarray.4*, *enters from the west*, the B array *from the north* and the C array from the southeast. Later the programmer will bind file names to the stream names to run the program.

This is the complete Poker program implementing the Kung-Leiserson algorithm, three pictures, one table and a piece of text. It is easy to read, and as we shall see later, it is easy to write. What makes the presentation clear and convenient is that the abstraction that we use when we communicate systolic algorithms between people, for example in textbooks, are very similar to the abstractions provided in the programming environment. This makes programming easy and convenient, thus fulfilling the objectives of the programming environment.

The translation of the original algorithm has been so complete as to be rather unrealistic, since the processes never halt. Though the program will run if we present the data files, wait briefly for the last good values to "drain" out, "kill" the program, and keep the output file, which may have some trailing zeros, it would be better to revise the program to terminate cleanly when the input data is exhausted. This will be done in the next section. To complete this section, the matter of synchronization must be addressed.

As was mentioned before Poker was designed to support a larger class of parallel algorithms than the systolic algorithms and so asynchronous communication was provided rather than synchronous communication. This does not affect the functionality of systolic programs because the basic data driven communication protocol, which blocks reads where data has not yet arrived at a processor assures that the systolic semantics are preserved, the right data gets combined. The main visible difference is that when one watches an asynchronous systolic array, the processors transmit data at apparently random times. But there is no real functional limitation to the facilities provided by Poker.

If one runs systolic algorithms on machines with asynchronous communication protocols, they will work, but there is a performance penalty because the protocol's "handshaking" incurs more overhead than does synchronous communication. It is possible to automatically restore a systolic program written with the asynchronous semantics of XX, to be synchronous again. This is accomplished by applying a program optimization technique called ***coordination*** [9]. In essence, a program's input and output statements are rearranged and idles are inserted, if needed, so that corresponding sends and receives between two processors are done at the same time based on a global clock. Though software is available for coordinating XX programs [10], it is not presently installed in Poker; thus all programs run asynchronously.

Improving the Poker Program

Having implemented a direct translation of a textbook algorithm, it will be useful to perform some minor changes to the program to make it more realistic and to illustrate additional language features. The modifications to be performed are (1) to remove the need to read in the C array, since it can be generated internally, (2) to avoid writing out A and B, and (3) to terminate the program when the input ends.

To begin, the I/O pads on the east and south sides of the array can be removed. This change is implemented by deleting the edges shown in the communication structure as shown in Figure 6; all other forms of the graph, for example the interconnections shown in the process assignment display and the port name assignment display, are derived or inherited from the general communication graph, and thus are changed automatically. The removal of the pads has the additional side effect of reducing the number of streams that must be defined. (See Figure 7.) This results in a somewhat more natural program in that all that remains are the two inputs and the one output band matrix.

Although the diagrams and the stream definitions are modified as a result of the change to the communication structure, other changes are not automatically made. For example, the port names around the east and south perimeter remain, and the *inner* process still writes to these ports. These changes must be done by the programmer.

There are two ways to change the processing definition. One is to modify the *inner* process so that when it is assigned to a processor on the east or south edge of the hex-mesh, that is, to a processor whose *i* or *j* index is maximal, then the process does not read a C value, nor does it write an A or B value (as appropriate, depending on which index is maximal). The required *inner* process would have an *if* statement before each of the three statements

```
   c <- Cin
Aout <- a
Bout <- b
```

testing to see whether the PE executing the code is a processor element with a maximal index and if so, bypassing the latter two statements and initializing to zero in the first case. Notice that because all processors are assigned the *inner* process with this solution, the process assignment of Figure 3 is unchanged.

The second way to change the process definition is to create different types of processes for the PEs boardering the east and south. This solution logically implies three types of

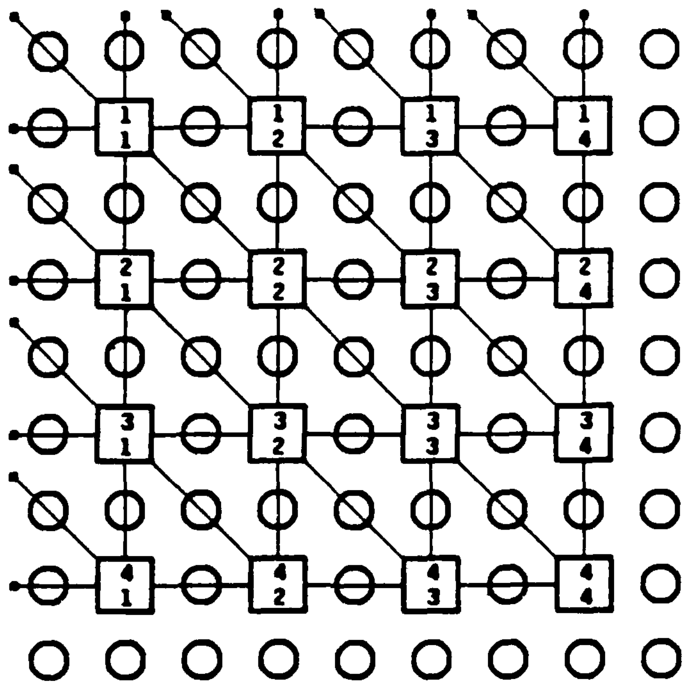


Figure 6. Revised communication structure

STREAM				DESTINATION				
PAD	NAME	INDEX	DIR.	PORT NAME	DIRECTION	CODE NAME	I	J
1	Barray	1	input	Bin	north	inner	1	1
2	Carray	4	output	Cout	northwest	inner	1	2
3	Barray	2	input	Bin	north	inner	1	2
4	Carray	3	output	Cout	northwest	inner	1	3
5	Barray	3	input	Bin	north	inner	1	3
6	Carray	2	output	Cout	northwest	east	1	4
7	Barray	4	input	Bin	north	east	1	4
8	Aarray	4	input	Ain	west	south	4	1
9	Carray	1	output	Cout	northwest	south	4	1
10	Aarray	3	input	Ain	west	inner	3	1
11	Carray	7	output	Cout	northwest	inner	3	1
12	Aarray	2	input	Ain	west	inner	2	1
13	Carray	6	output	Cout	northwest	inner	2	1
14	Aarray	1	input	Ain	west	inner	1	1
15	Carray	5	output	Cout	northwest	inner	1	1

Figure 7. Revised stream assignment

programs: Those of the eastern edge, except the southeast corner, are like *inner* except that they do not read C nor write A; those of the southern edge, except the southeast corner, are like *inner* except that they do not read C nor write B; the process of the southeast corner does not read C, nor write either A or B. These versions, shown in Figure 8, require the modification to the process assignment specification shown in Figure 9.

The first solution is easier to program while the second solution has better performance because the processor elements are not wasting their time repeatedly testing a predicate that never changes. The second solution is preferred, of course, though they are equivalent in the presence of an optimizing compiler that performs constant folding and dead code elimination.

The processes of Figure 8 also reveal the strategy for terminating the program when the input is exhausted. The technique uses a special input stream terminator, *EOS*, mnemonic for end of stream. Each process in the array will be halted by the arrival of an EOS symbol although different processes will be halted by different streams: The eastern PEs will be terminated by the end of the B stream, the southern PEs by the end of the A stream, the southeastern PEs can be terminated by either stream - stream A has been selected - and the remainder of the PEs are terminated by EOS on the C stream in order that the last good values computed drain out of the array. The *dataavail* predicate tests to be sure that data has arrived before reading; this is necessary to prevent the read statement from reading passed the EOS. All processors will halt often passing along the EOS token that caused them to break out of the loop. The output file will also be terminated by an EOS.

Figure 10 shows a revised version of the port names assignment. This is not strictly necessary. The port names assignment of Figure 4 could serve because Poker permits port labels with no corresponding communication paths as long as the port is not written to or read from. This condition obviously holds for the process definition of Figure 8. We chose to revise it because the new version has the property that all and only the connected ports are labelled; this is a property that Poker will test for us, if we wish, as a correctness check.

The revised program is not significantly more complex than its predecessor and it is about as easy to read.

Poker Programming Environment

A discussion of the main constituents of a Poker program and an explanation of the effect of various modifications on it are adequate preparation for presenting the structure and organization of the Poker environment [11].

```
code inner;
trace a,b,c;
ports Ain, Bin, Cin,
      Aout, Bout, Cout;
begin
  real a,b,c;
  Cout <- 0.0;
  Bout <- 0.0;
  Aout <- 0.0;
  while ~isEOS(Cin) do
    if dataavail(Cin)
      then
      begin
        c <- Cin;
        b <- Bin;
        a <- Ain;
        c := c + a*b;
        Cout <- c;
        Bout <- b;
        Aout <- a
      end;
  Cout <- EOS
end.
```

```
code east;
trace a,b,c;
ports Ain, Bin,
      Bout, Cout;
begin
  real a,b,c;
  Cout <- 0.0;
  Bout <- 0.0;
  while ~isEOS(Bin) do
    if dataavail(Bin)
      then
      begin
        b <- Bin;
        a <- Ain;
        c := 0.0;
        c := c + a*b;
        Cout <- c;
        Bout <- b
      end;
  Cout <- EOS;
  Bout <- EOS
end.
```

```
code south;
trace a,b,c;
ports Ain, Bin,
      Aout, Cout;
begin
  real a,b,c;
  Cout <- 0.0;
  Aout <- 0.0;
  while ~isEOS(Ain) do
    if dataavail(Ain)
      then
      begin
        a <- Ain;
        b <- Bin;
        c := 0.0;
        c := c + a*b;
        Cout <- c;
        Aout <- a
      end;
  Cout <- EOS;
  Aout <- EOS
end.
```

```
code se;
trace a,b,c;
ports Ain, Bin, Cout;
begin
  real a,b,c;
  Cout <- 0.0;
  while ~isEOS(Ain) do
    if dataavail(Ain)
      then
      begin
        a <- Ain;
        b <- Bin;
        c := 0.0;
        c := c + a*b;
        Cout <- c
      end;
  Cout <- EOS;
end.
```

Figure 8. Revised process definitions

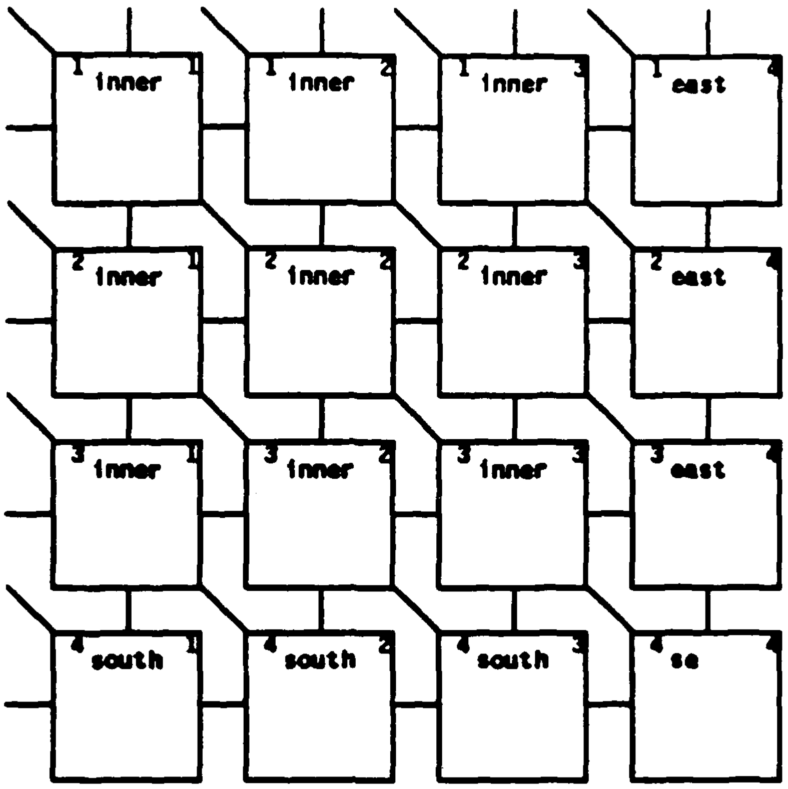
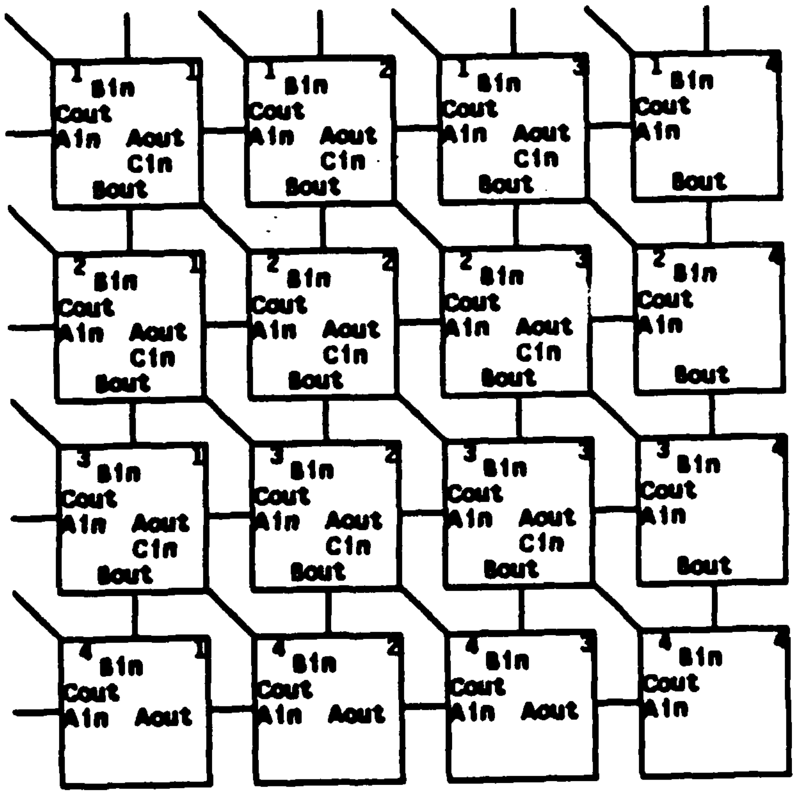


Figure 9. Revised process assignment



1 Bin Cout Ain Aout Cin Bout 1
1 Bin Cout Ain Aout Cin Bout 2
1 Bin Cout Ain Aout Cin Bout 3
1 Bin Cout Ain Bout 4
2 Bin Cout Ain Aout Cin Bout 1
2 Bin Cout Ain Aout Cin Bout 2
2 Bin Cout Ain Aout Cin Bout 3
2 Bin Cout Ain Bout 4
3 Bin Cout Ain Aout Cin Bout 1
3 Bin Cout Ain Aout Cin Bout 2
3 Bin Cout Ain Aout Cin Bout 3
3 Bin Cout Ain Bout 4
4 Bin Cout Ain Aout 1
4 Bin Cout Ain Aout 2
4 Bin Cout Ain Aout 3
4 Bin Cout Ain 4

Figure 10. Revised portnames assignment

Poker is an interactive graphic system using two displays, one of which is a high resolution bitmapped display. This terminal, called the primary display, is used for all programming activity except the creation of the process definitions which are developed on the secondary display using a standard editor. Figure 11 shows the format of a typical primary display. Notice that the information shown in Figures 3-5 is also displayed at the bottom of the screen in a similar format.

The Poker environment stores the program constituents of Figures 1-5 in a database. The creation and modification of this information is organized around a set of *views*. A view generally displays one of the program constituents, for example, the process assignment, and provides interactive commands for graphically editing the program constituent. The available views are:

Switch Setting View. Displays the communication structure; the user performs such activities as drawing lines between boxes to establish communication channels, using a mouse or cursor keys.

Code Names View. Displays the process assignment information; the user moves among the (processor) boxes, entering the name of the process to be assigned to the processor as well as any parameters it might have.

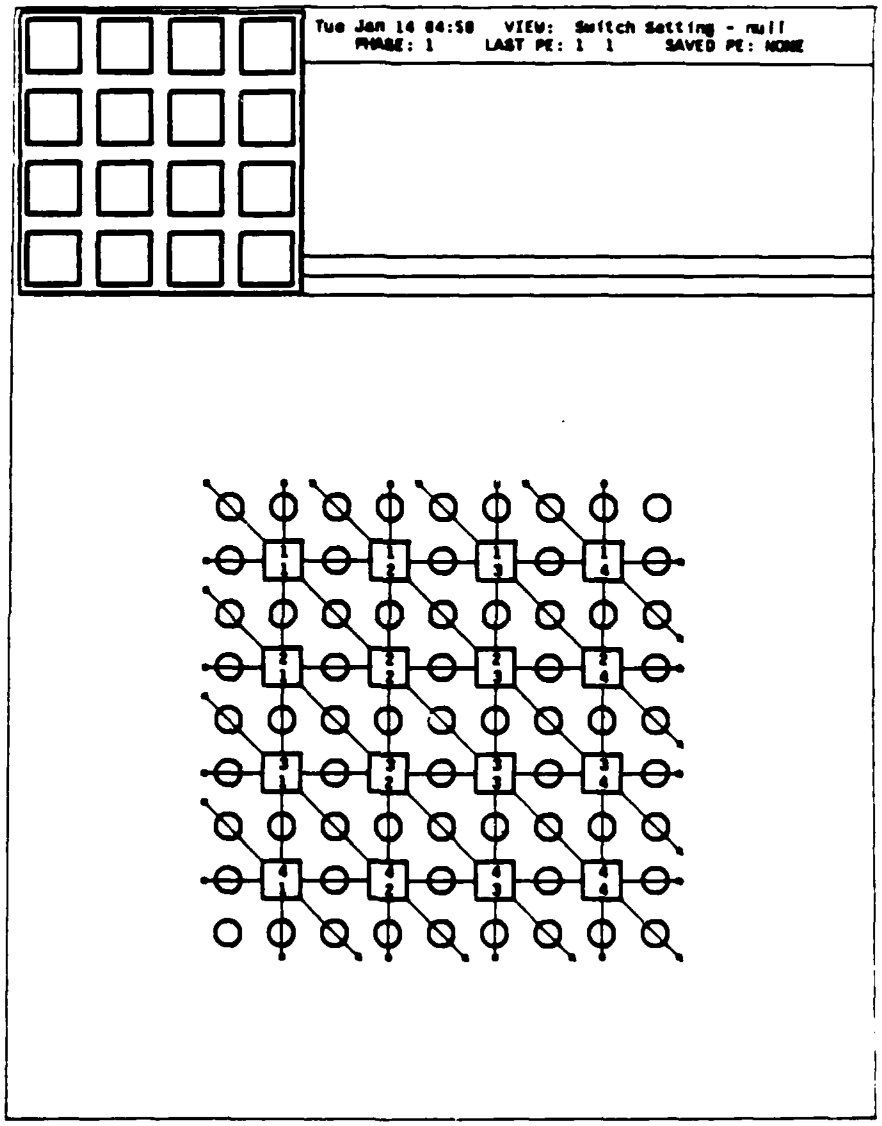
Port Names View. Displays the port name assignment information; the user moves among the (processor) boxes labelling any edges connected to it.

I/O Names View. Displays the stream name assignment information together with a schematic of the communication structure, see Figure 12; the user enters symbolic names and indexes for the streams as well as defining whether they are input or output.

Command Request View. Provides the user with the ability to compile, assemble, coordinate [9, 10] link, and load the program information of the database; the display shows the progress of these transformations.

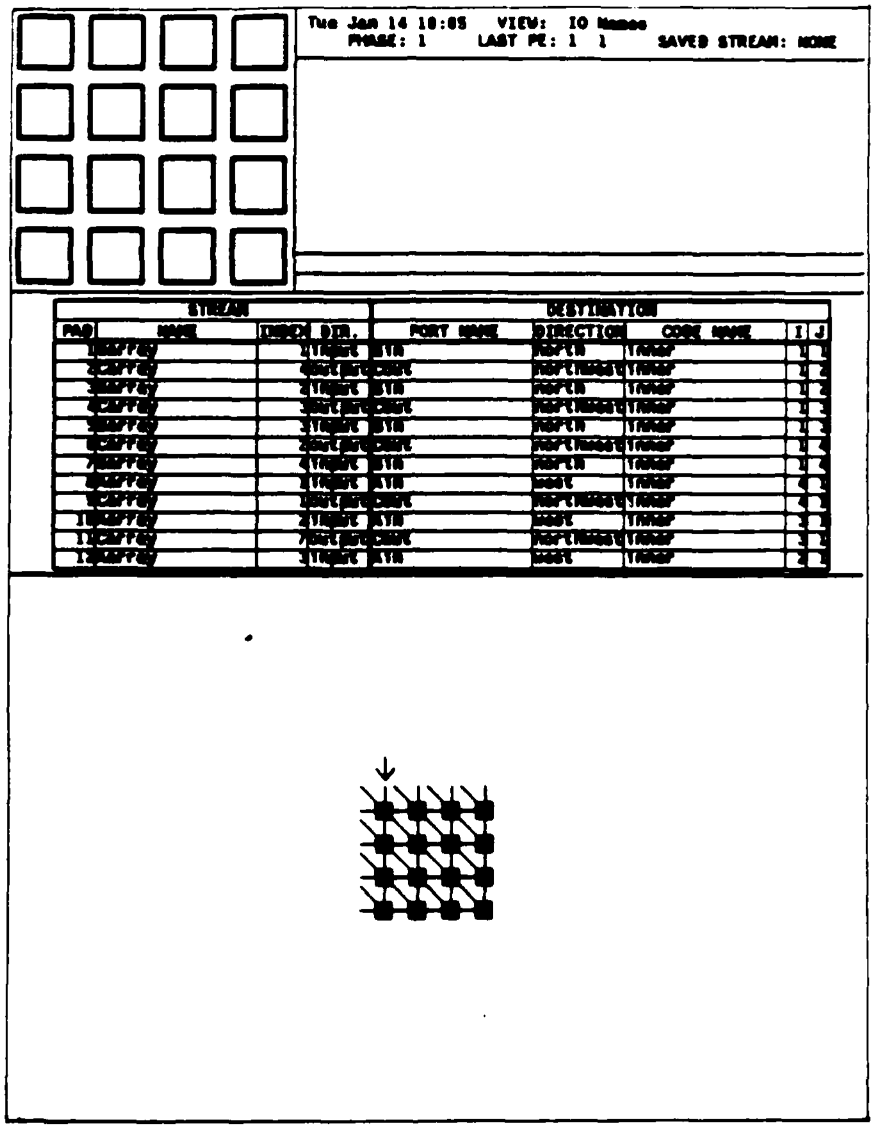
Trace View. Displays the progress of the program execution; the user initiates execution, "single steps execution", checkpoints execution, etc. as well as watching the consecutive values of the traced variables as the execution proceeds; the source of the execution can be either a physical machine or an emulator.

CHiP Parameters View. Displays the current architecture being programmed; the user edits the parameters to change the machine description; for example, one can change the number of processors in the processor array.



```
Tue Jan 14 04:50   VIEW:  Switch Setting - null
     PHASE: 1      LAST PE: 1  1     SAVED PE: NONE
```

Figure 11. Sample of a typical Primary Poker Display



Tue Jan 14 10:05   VIEW: IO Names
PHASE: 1   LAST PE: 1 1   SAVED STREAM: NONE

STREAM				DESTINATION				
PAD	NAME	INDEX	DIR.	PORT NAME	DIRECTION	CODE NAME	I	J
1	aArray	1	input	ain	north	inner	1	1
2	cArray	4	output	cout	northwest	inner	1	2
3	aArray	2	input	ain	north	inner	1	2
4	cArray	3	output	cout	northwest	inner	1	3
5	aArray	3	input	ain	north	inner	1	3
6	cArray	2	output	cout	northwest	inner	1	4
7	aArray	4	input	ain	north	inner	1	4
8	aArray	1	input	ain	west	inner	4	1
9	cArray	1	output	cout	northwest	inner	4	1
10	aArray	2	input	ain	west	inner	3	1
11	cArray	7	output	cout	northwest	inner	3	2
12	aArray	3	input	ain	west	inner	2	1

Figure 12. Example of the I/O Names View

Each view provides many more facilities than have been suggested here, and describing them thoroughly would lengthen this paper to a programming manual [11]. It should be noted, however, that in addition to the special operations required for each view, there are screen management commands, library facilities, online help facilities, file transfer facilities, etc. so that Poker is a *completely self-contained environment.*

A Poker Programming Session

Poker is extremely interactive – each keystroke generally causes some immediate action to take place. Furthermore, because the display changes with each keystroke or so, it is very dynamic. Finally, except for the text of the process definitions, Poker programs are not stored in symbolic form; they are stored in a database that has no meaningful printed form. [The diagrams for Poker papers are produced by capturing the bits from the bitmapped display.] These characteristics of Poker make textual description quite unsatisfactory for capturing the "feel" of the system. Nevertheless, it is instructive to step through the main activities performed in development of a Poker program.

The session begins with the programmer entering Poker from UNIX^{TM}; it is possible to remain in Poker until the program is written, debugged and run. If this is a new program, the programmer may want to specify, using the CHiP Parameter View, a particular machine configuration different from the default and more closely matching the problem at hand. This activity of "changing the architecture" may seem peculiar: If a programmer has a particular machine to use, why not use its characteristics in programming the problem, since they are presumably the default characteristics? One answer is that it is often much faster to get a problem working on a small array rather than on a large one because the emulator, a component of Poker, provides fast convenient tracing of small arrays. Enlarging the solution is usually straight-forward and simple, making this an effective programming style. (If this is a program that has been previously worked on, the programmer is presented with the exact state in force at the conclusion of the previous session. The following discussion presumes a new program.)

Although one can start programming in almost any view, most programmers begin by defining the communication structure using the Switch Setting View. The programmer is shown an array of circles (which can be ignored here) and boxes and the task is to draw lines with a mouse or cursor keys to connect the boxes together. These lines establish the bidirectional datapaths to be used by the processor elements for interprocessor communication. Like the architectural specification changes mentioned above, this programmer definition of the interconnection structure seems very peculiar: If a programmer is using a systolic array or other nonshared memory, nonconfigurable computer, there is only one physical communication structure. Why isn't it the default communicating structure? It could be, and as long as it matches the communication structure needed by the algorithm, it should be. But often the logical communication structure used by the algorithm is different from the physical structure of the machine, and it is then that the programmer is advised to work initially with the logical structure. The program can be written and tested (on the simulator) based only on the logical structure. The use of the appropriate logical structure simplifies the programming and reduces the complications of the debugging; it is a natural abstraction to reduce the complexity of synchronization operations. After the program is running, it can be revised either by the programmer or by a routine of the Command Request View, to utilize the physical communication structure directly. The choice depends on the peculiarities of the target machine and the availability of automatic mapping software [12].

Having completed the interconnection structure, programmers often move to the secondary terminal and define one or more processes. This activity uses a standard editor and amounts to standard sequential programming in an algebraic language such as C, PASCAL, or the specialized PASCAL-like language, XX. Poker can be used with just a single terminal, but the advantage of using two is important: As the programmer develops the program on the secondary display and thereby creates process names, port names, formal parameters, etc., it is convenient to assign the information immediately to the appropriate position in the various Poker views. This can be accomplished without diminishing the visable information when two displays are in use simply by moving from one to the other.

When the programmer moves to the Code Names View to define the process assignment, the display shows the collection of empty boxes connected in whatever arrangement was defined in the Switch Setting View, if any. By using the mouse or cursor keys to move from box to box and from line to line within a box, the programmer can assign process names to the processors and actual parameters to the processes. Notice that there is a simple protocol of perhaps a half dozen keystrokes to store the same information into one or more rows or columns (including all processors). In this way, the regularity so often found in systolic arrays can be exploited to simplify the programming task without restricting the flexibility of the system.

The Port Names View is much the same as the Code Names View except that each process box is divided into eight windows corresponding to the eight compass points. The programmer uses the cursor motions to move to those windows corresponding to an edge connecting to the box. As before, computer assistance is available for automatically assigning information that is consistent from processor to processor.

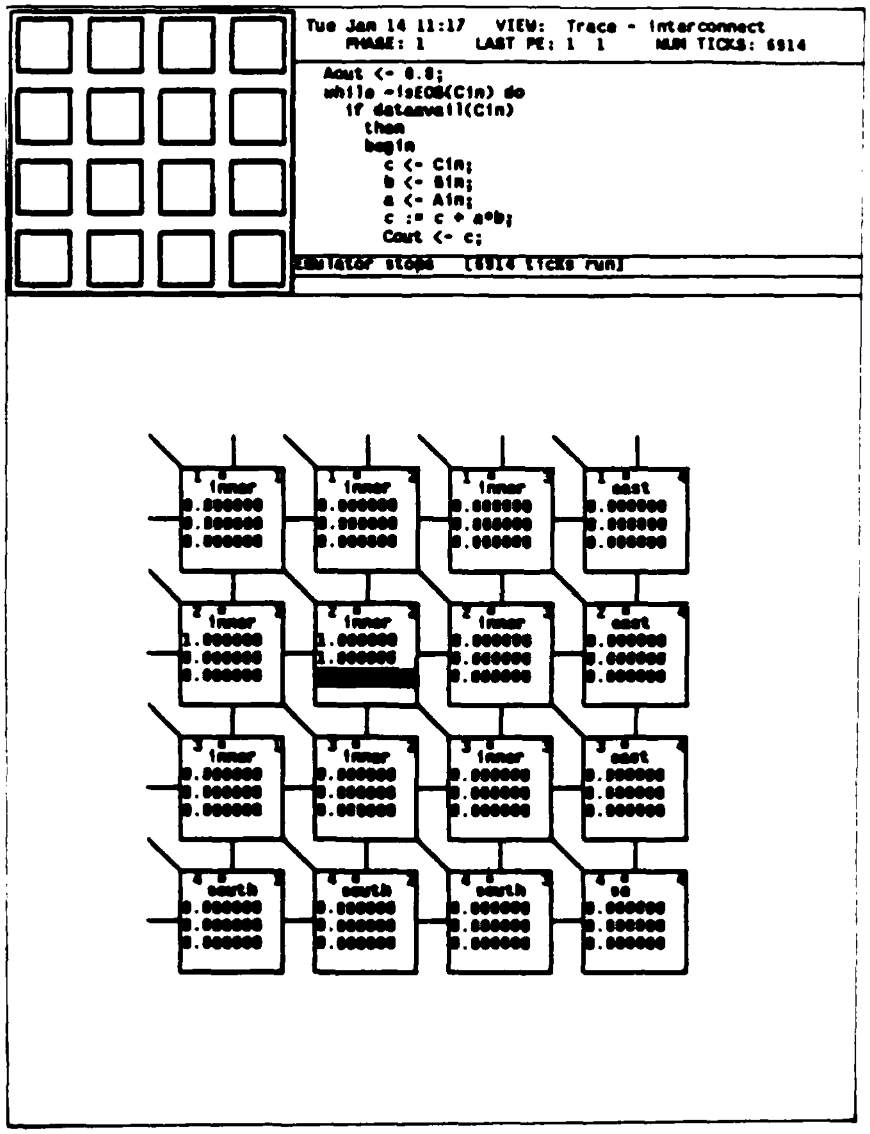
The I/O Names View shares properties with the preceding views – moving from window to window, assigning names to program objects, computer assistance for repetitive entires, etc. - but it is unique in several respects too. The programmer is shown a listing of all I/O pads used in the program (numbered from the northwest corner), with the processor and port they connect to and, if defined, the port name of that port and the process name assigned to that processor. This is not a graphical way to convey or request data. But as a heuristic to remind the programmer which stream is which, a schematic diagram of the interconnection graph is given on the bottom of the screen and an arrow points to that I/O pad to whose entry the cursor presently points. (See Figure 12.)

Typically, the programmer moves between views frequently correcting and revising the program as changes of one part mandate changes in another. When the program is finished, the programmer will move to the Command Request View to convert the program to object form. The various activities can be done selectively or batched in a single keystroke "make" command. The only remaining task is to bind file names to the stream names so as to define the data the program is to process. Special commands make the association. In a production situation where the program is already written and compiled, etc., the "operator" who is to run the program on many sets of data will operate from the Command Request View. Successive data sets will be bound to the stream names and the program will be invoked. If the set of data sets is routinely the same, then the script facilities are used and the operator need not even be present.

Assuming the programmer wants to watch the program run, and has specified some of the names of variables to be traced in the preambles of the programs, the Trace View will begin with the process boxes shown with the names of the assigned process and the initial values of the traced variables. The programmer has a variety of ways to control the running of the program. As the progress of the processing causes variables of a process to change values, the new values are shown on the screen, highlighted together with the current clock time. Figure 13 shows this display at the moment the first scalar inner product computation is completed (see highlight) in PE 2.2. The program can be checkpointed, restarted, single-stepped, executed until a variable changes value, etc. If the program is faulty in any way, the programmer can return to one of the views to check or change an entry; otherwise the program can be stored as a unit for future use simply by exiting.

Summary

A programming environment for systolic arrays would provide complete facilities to support all aspects of systolic program development, debugging and execution based on abstractions that are convenient and perspicuous. Poker, though developed for a more general class of algorithms, illustrates to a substantial degree, the form these facilities might take and how the abstractions might be implemented.



```
Tue Jan 14 11:17   VIEW:  Trace - interconnect
    PHASE: 1      LAST PE: 1  1     NUM TICKS: 6914

  Aout <- 0.0;
  while ~isEOS(Cin) do
    if dataavail(Cin)
      then
      begin
        c <- Cin;
        b <- Bin;
        a <- Ain;
        c := c + a*b;
        Cout <- c;
Emulator stops   [6914 ticks run]
```

Figure 13. The Trace View

Specifically, Poker illustrates comprehensive support for everything from drawing pictures of communication graphs to library facilities; the programmer would need no other interface to the systolic array with such facilities. The program form used by the system is nonstandard, being based on three types of diagrams, a table, and segments of text. Nevertheless, this program form is more convenient to use than text files and more comprehensible, because it utilizes graphics more and symbolic presentations less. To support systolic computation completely one must replace the asynchronous communication style of Poker with a synchronous execution mode. This modification would only really affect the programming language(s) used for the processor elements and would not change the overall approach to parallel programming.

References

1. Lawrence Snyder
   Parallel Programming and the Poker Programming Environment
   Computer 17(7):27-36, July 1984.

2. Lawrence Snyder
   Configurable, Highly Parallel (CHiP) Approach to Signal Processing Applications
   In *Proceedings of Technical Symposium East '82*, Society of Photo-Optical and Instrumentation Engineers, pp. 8-16, 1982.

3. Yasunori Dohi, Allan L. Fisher, H. T. Kung and Louis M. Monier
   The Programmable Systolic Chip: Project Overview
   In L. Snyder, L. H. Jamieson, D. B. Gannon and H. J. Siegel, editors, *Algorithmically Specialized Parallel Computers*, Academic Press, 1985.

4. Keith Bromley, J. S. Symanski, J. M Speiser and H. J. Whitehouse
   Systolic Array Processor Developments
   In H. T. Kung, Bob Sproull and Guy Steele, editors, *VLSI Systems and Computations*, Computer Science Press, pp. 273-284, 1981.

5. Lawrence Snyder
   Introduction to the Configurable, Highly Parallel Computer
   Computer 15(1):47-56, January 1982.

6. Alejandro Kapauan, J. Timothy Field, Dennis B. Gannon and Lawrence Snyder
   The Pringle Parallel Computer
   In *Proceedings of the 11th International Symposium on Computer Architecture* pp. 12-20, IEEE, 1984.

7. Charles E. Seitz
Cosmic Cube
CACM 28(1):22-33, 1985.

8. Carver Mead and Lynn Conway
Introduction to VLSI Systems
Addison-Wesley, 1980.

9. Janice E. Cuny and Lawrence Snyder
Compilation of Data-driven Programs for Synchronous Execution
In *Proceedings of the 10th ACM Symposium on the Principles of Programming Languages* pp. 197-202, 1983.

10. Duane A. Bailey, Janice E. Cuny and Bruce B. MacLeod
Parallel Code Optimization to Reduce Communication Overhead
Technical Report, COINS Department, University of Massachusetts, 1985.

11. Lawrence Snyder
Poker (3.1): A Programmer's Reference Guide
Technical Report 85-09-03, Department of Computer Science, University of Washington, 1985.

12. Francine Berman, M. Goodrich, C. Koelbel, W. Robison and K. Showell
Prep-P: A Mapping Preprocessor for CHiP Computers
In *Proceedings of the 1985 International Conference on Parallel Processing* Pheasant Run, Illinois.